OBJECT POINTERS
MODULE 6

Pointers to Objects

- Objects are stored in memory, so pointers can point to objects just as they can to variables of basic types.
- The `new` Operator:
  - The `new` operator allocates memory of a specific size from the operating system and returns a pointer to its starting point. If it is unable to find space, in returns a 0 pointer.
  - When you use `new` with objects, it does **not only** allocates memory for the object, it also creates the object in the sense of invoking the object's constructor.
  - This guarantees that the object is correctly initialized, which is vital for avoiding programming errors.

Pointers to Objects

The delete Operator

> To ensure safe and efficient use of memory, the new operator is matched by a corresponding delete operator that releases the memory back to the operating system.

> If you create an array with new Type[ ];, you need the brackets when you delete it:

```
int * ptr = new int[10];
int []ptr = {4,8,15,16,23,42} ;
delete [ ] ptr;
```

Don't forget the brackets when deleting arrays of objects. Using them ensures that all the members of the array are deleted and that the destructor is called for each one. If you forget the brackets, only the first element of the array will be deleted.

Example

```
class String {
   int size;
   char *contents;
 public:
   String();
   String(const char *);
   String(const String &);
   const String& operator=(const String &);
   void print() const ;
   ~String();
};
```

Example

```
int main()  {
    String *sptr = new String[3];
    String s1("String_1");
    String s2("String_2");
    *sptr = s1;
    sptr[0] = s1;
    *(sptr + 1) = s2;
    sptr[1] = s2;
    sptr->print();
    (sptr+1)->print();
    sptr[1].print();
    delete[] sptr;
    return 0;
}
```

Linked List of Objects

> A class may contain a pointer to objects of its type.

> This pointer can be used to build a chain of objects, a linked list.

```
class Teacher {
    friend class ListOfTeacher;
    string name;
    int age, numOfStudents;
    Teacher * next;
 public:
    Teacher(const string &, int, int);
    void print() const;
    const string& getName() const {
         return name;}
   ~Teacher()
};
```

Linked List of Objects

```
// linked list for teachers
class ListOfTeacher {
    Teacher *head;
  public:
    ListOfTeacher(){head=0;}
    bool append(const string &,int,int);
    bool del(const string &);
    void print() const ;
    ~ListOfTeacher();
};
```

Linked List of Objects

- In the previous example the **Teacher** class must have a pointer to the next object and the list class must be declared as a friend, to enable users of this class building linked lists.
- If this class is written by the same group then it is possible to put such a pointer in the class.
- But usually programmers use ready classes, written by other groups, for example classes from libraries.
- These classes may not have a next pointer.
- To build linked lists of such ready classes the programmer have to define a node class.
- Each object of the node class will hold the addresses of an element of the list.

Linked List of Objects

```
class TeacherNode {
    friend class ListOfTeacher;
    Teacher * element;
    TeacherNode * next;     // next node
    TeacherNode(const string &, int, int);
    ~TeacherNode();
};
TeacherNode::TeacherNode(const string & name,
                         int age,int numOfStudents){
    element = new Teacher(name,age,numOfStudents);
    next = 0L;
}
TeacherNode::~TeacherNode(){
   delete element;
}
```

Pointers and Inheritance

- If a class **Derived** has a public base class **Base**, then a pointer to **Derived** can be assigned to a variable of type pointer to **Base** without use of explicit type conversion.
- A pointer to **Base** can carry the address of an object of **Derived**.
- The opposite conversion, for pointer to **Base** to pointer to **Derived**, must be explicit.
- For example, a pointer to **Teacher** can point to objects of **Teacher** and to objects of **Principal**.
  - A principal is a teacher, but a ***teacher*** is ***not always*** a ***principal***.

Pointers and Inheritance

```
class Base{
};
class Derived : public Base {
};
Derived d;
// implicit conversion
Base *bp = &d;
// ERROR! Base is not Derived
Derived *dp = bp;
// explicit conversion
dp = static_cast<Derived *>(bp);
```

Pointers and Inheritance

- If the class **Base** is a private base of **Derived**, then the implicit conversion of a **Derived*** to **Base*** would not be done.
- Because, in this case a public member of **Base** can be accessed through a pointer to **Base** but not through a pointer to **Derived**.

Pointers and Inheritance

```
class Base {
    int m1;
  public:
    int m2;     // m2 is a public member of Base
};
class Derived : private Base {
   // m2 is not a public member of Derived
    :
};
Derived d;
d.m2 = 5; // ERROR!
Base *bp = &d; // ERROR!
```

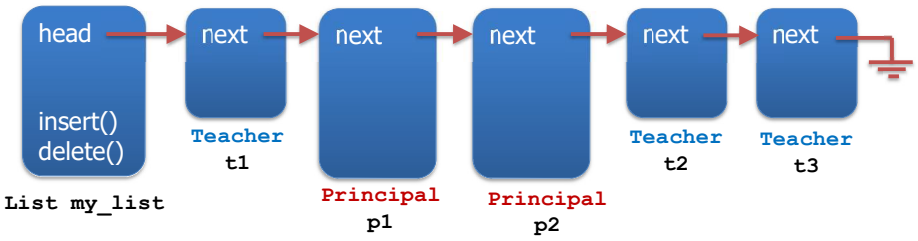
Heterogeneous Linked Lists

> Using the inheritance and pointers, heterogeneous linked lists can be created.

> A list specified in terms of pointers to a base class can hold objects of any class derived from this base class.

> We will discuss heterogeneous lists again, after we have learnt polymorphism.

> **Example:** A list of teachers and principals



Class `auto_ptr`

> The **`auto_ptr`** type is provided by the C++ standard library as a kind of a smart pointer that helps to avoid resource leaks when exceptions are thrown.

> There are several useful smart pointer types.

> This class is smart with respect to only one certain kind of problem.

> For other kinds of problems, type **`auto_ptr`** does not help.

– So, be careful!

Motivation of Class `auto_ptr`

> Functions often operate in the following way

1. Acquire some resources.
2. Perform some operations.
3. Free the acquired resources.

> Example:

```
void f() {
  ClassA* ptr = new ClassA;
  ... //perform some operations
  delete ptr; //clean up
}
```

> This function is a source of trouble.

Motivation of Class `auto_ptr`

> An exception would exit the function immediately ***without calling*** the `delete` statement at the end of the function.

> The result would be a **memory leak** or, more generally, a resource leak.

Motivation of Class `auto_ptr`

> Solution:

```
void f() {
  ClassA* ptr = new ClassA;
     try {
       ... //perform some operations
     }
     catch (...) { //for any exception
        delete ptr; //clean up
        throw; //rethrow the exception
     }
}
```

> **`auto_ptr`** was designed to be such a kind of smart pointer.

Use of **auto_ptr**

```
//header file for auto_ptr
#include <memory>
void f() {
   //create and initialize an auto_ptr
   std::auto_ptr<ClassA> ptr(new ClassA);
   ... //perform some operations
}
```

Transfer of Ownership by **auto_ptr**

```
//initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);
//copy the auto_ptr
//transfers ownership from ptr1 to ptr2
std::auto_ptr<ClassA> ptr2(ptr1);
```

Transfer of Ownership by **auto_ptr**

```
//initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);
//create another auto_ptr
std::auto_ptr<ClassA> ptr2;
//assign the auto_ptr
//transfers ownership from ptr1 to ptr2
ptr2 = ptr1;
```

Transfer of Ownership by **auto_ptr**

> If **ptr2** owned an object before an assignment, delete is called for that object:

```
//initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);
//initialize another auto_ptr with new obj.
std::auto_ptr<ClassA> ptr2(new ClassA);
// assign the auto_ptr
// delete object owned by ptr2
// transfers ownership from ptr1 to ptr2
ptr2 = ptr1;
```

Be careful!

> To assign a new value to an **auto_ptr**, this new value must be an **auto_ptr**.

> You can't assign an ordinary pointer:

```
//create an auto_ptr
std::auto_ptr<ClassA> ptr;
ptr = new ClassA; //ERROR
// OK, delete old object
// and own new
ptr = std::auto_ptr<ClassA>(new ClassA);
```

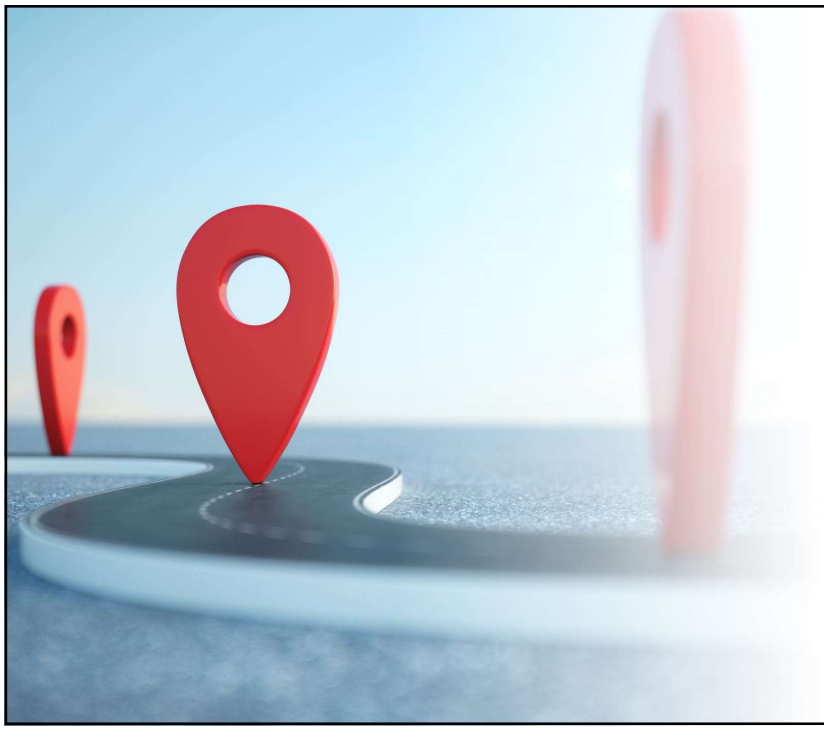
Functions and **auto_ptr**

> When an **auto_ptr** is returned, ownership of the returned value gets transferred to the calling function.

```
std::auto_ptr<ClassA> f() {
  std::auto_ptr<ClassA> ptr(new ClassA);
  //ptr owns the new object
  ...
  //transfer ownership to calling function
  return ptr;
}
```

Functions and **auto_ptr**

```
void g(){
  std::auto_ptr<ClassA> p;
  for (int i=0; i<10; ++i) {
    //p gets ownership of the returned object
    p = f();
    //previously returned object gets deleted
    ...
  }
} //last-owned object of p gets deleted
```



SMART POINTERS

Smart Pointers

> Since C++11, the C++ standard library provides two types of smart pointer:

1. Class **`shared_ptr`** for a pointer that implements the concept of *shared ownership*.
   - Multiple smart pointers can refer to the same object so that the object and its associated resources get released whenever the last reference to it gets destroyed.
   - To perform this task in more complicated scenarios, helper classes, such as **`weak_ptr`**, **`bad_weak_ptr`** are provided.

Smart Pointers

> Since C++11, the C++ standard library provides two types of smart pointer:

2. Class **`unique_ptr`** for a pointer that implements the concept of ***exclusive ownership*** or ***strict ownership***.
   - This pointer ensures that only one smart pointer can refer to this object at a time. However, you can transfer ownership.
   - This pointer is especially useful for avoiding resource leaks, such as missing calls of delete after or while an object gets created with new and an exception occurred.

Class `shared_ptr`

- Almost every nontrivial program needs the ability to use or deal with objects at multiple places at the same time.
- Thus, you have to “refer” to an object from multiple places in your program.
- Although the language provides references and pointers, this is not enough, because you often have to ensure that when the last reference to an object gets deleted, the object itself gets deleted, which might require some cleanup operations, such as freeing memory or releasing a resource.
- So we need a semantics of “cleanup when the object is nowhere used anymore.” Class **`shared_ptr`** provides this semantics of *shared ownership*

Example (1/3)

```
#include <iostream>   #include <string>
#include <vector>
#include <memory>
using namespace std;
int main(){
    shared_ptr<string> pJack(new string("jack"),
              [](string* p){
                   cout << "Deleting " << *p << endl;
                   delete p;
              });
    shared_ptr<string> pKate(new string("kate"));
    (*pJack)[0]='J';
    pKate->replace(0,1,"K");
    cout << *pJack << endl;
    cout << *pKate << endl;
```

Example (2/3)

```
vector<shared_ptr<string>> v;
v.push_back(pKate);    v.push_back(pKate);
v.push_back(pJack);    v.push_back(pJack);
v.push_back(pKate);    v.push_back(pJack);
cout << "for loop:" << endl ;
for (auto& p : v ){
    cout << *p << endl ;
}
cout << "use_count(): "
     << v[1].use_count() << endl ;
pJack=nullptr;
v.resize(2);
cout << "for loop:" << endl ;
```

Example (3/3)

```
        for (auto& p : v ){
            cout << *p << endl ;
        }
        cout << "use_count(): "
             << v[1].use_count() << endl ;
    }
```